Computer Science I: Quiz 1 (Programs 4, 6) Solutions

1) (10 pts) In program #4 (maze), the input was a grid of size R x C, indicating a starting square, blocked squares and squares to travel through. The goal of the assignment was to determine the shortest number of moves to get to the edge of the grid (in the assignment marked by the '~' character.) In terms of R and C, with proof, determine the run time of the posted solution. Please clearly justify your answer.

Solution

The key work of the breadth first search is to process grid squares via the queue. Due to the used/distance array, no square from the grid gets enqueued and dequeued more than once. Thus the while loop runs at most RC times. Inside the while loop, we go through all adjacent grid squares of the ones being processed (this is the for loop that goes through the DX/DY arrays). This loop runs a constant (4) number of times. Thus, the whole run time of the posted solution in terms of R and C is **O(RC).**

Run time = **O(RC).**

Grading: 5 pts for the correct answer (this is all or nothing), 5 pts for the reason - 3 pts for arguing that the while loop can't run more than RC times, 2 pts for arguing that the contents of a single while loop iteration run in constant time. An incorrect answer can get in between 0 and 5 points inclusive.

2) (5 pts) In the posted solution to program #4 (maze), the queue stored each grid location (ordered pair) as a single integer. When necessary, a single integer that was dequeued was "converted" into two separate values, x and y, representing the row and column of the desired square in the grid. Show that you understand this conversion by doing it for the following case: Let R = 20, the number of rows in the grid and C = 30, the number of columns in the grid. What is the row and column of the grid square designated by the integer 379? (Yes, you have to do some arithmetic on your own…) Note: the very first row is row 0 and very first column is column 0.

Solution

The key to this calculation is to use integer division and mod as follows with the ***column*** size of the grid. (To see this, note that row 0 contains 30 elements, so the value of the first item on row 1 would be 30.)

row = 379/30 = 12
col = 379%30 = 19

Row = **12** Column = **19**

Grading: 2 pts out of 5 if they used 20 for mod and div, 4 pts out of 5 if they had the correct mod and div written but did the arithmetic somewhere incorrectly, 3 pts out of 5 if they switch where to do the mod and where to do the div.

3) (5 pts) In the posted solution for program #4 (maze), a queue is used. Each item in the queue represents a reachable grid square from the starting point. For each of these grid squares, the solution calculates the shortest distance from the starting point to the grid square. At any given point in time while the code is running, what is the largest difference between the distance from the starting point to two different grid squares that are simultaneously in the queue? Give proof of your answer.

Solution

The answer is just 1. Notice that when we dequeue the starting square from the queue, we add all grid squares that are distance 1 away from it. In general, whenever we dequeue any item that is a distance d from the start, the items we add to the back of the queue are only distance d+1 from the start. Since we must first dequeue an item that is distance d+1 from the start before we add any items that are distance d+2 from start to the queue, this means that while there are still items that are only distance d from the start left in the queue, no item that is distance d+1 from start has been dequeued yet. This is because the breadth first search first processes everything d units away before it gets to anything d+1 units away. Thus, there's never items in the queue simultaneously who have a difference of distance from the start square of more than 1.
Largest difference = **1.**

Grading: 3 pts for the answer, all or nothing, 2 pts for the explanation - it can be much shorter than mine and not quite as specific.

4) (5 pts) The function below is in the extra credit version of the posted solution to program #6 (boggle). Comments have been removed and the name of the function has been changed. What purpose does this function serve and why was it necessary in the context of this extra credit solution?

```
void function(trie* dictionary) {
    if (dictionary != NULL) {
        dictionary->inGrid = 0;
        int i;
        for (i=0; i<26; i++)
            function(dictionary->nextLetter[i]);
    }
}
```

Solution

The method the posted extra credit solution uses to only print words once as follows: once a valid word is discovered in the grid, rather than printing it, the word is flagged inside the trie in a separate variable in the trie struct, inGrid. Even if a word appears multiple times in a grid, this variable in the trie struct is marked 1 multiple times, which doesn't affect how many times it prints. After a single input case is solved, a preorder type traversal is done on the trie, printing out each word associated with nodes with the variable inGrid == 1. After this printing is done, to prepare for the next input case, these inGrid values have to be reset to 0. That is the purpose of this function, to clear out the inGrid field of every node in the trie so that a new input case can be processed.

Grading: 3 pts for saying "setting all inGrid fields to 0", 2 pts for explaining why.

5) (5 pts) Consider the following solveIt function written below. It has been modified from the solution posted for program #6 (boggle). (Note: in addition, the name of the trie was modified from dictionary to words so lines didn't wrap in the Word doc of this quiz.) If we were to use this function instead of the one in the posted solution, how does it assume the rules of the game have changed?

```
void solveIt(char prefix[], int curX, int curY, char board[][SIZE],
trie* words) {

  int curLen = strlen(prefix);
  if (isWord(prefix, words))printf("%s\n", prefix);
  if (!isPrefix(prefix, words)) return;

  int i;
  for (i=0; i<NUMDIR; i++) {
    if (inbounds(curX+DX[i], curY+DY[i])) {
      prefix[curLen] = board[curX+DX[i]][curY+DY[i]];
      prefix[curLen+1] = '\0';
      solveIt(prefix, used, curX+DX[i], curY+DY[i], board, words);
      prefix[curLen] = '\0';
    }
  }
}
```

Solution

This solution allows the user to reuse individual grid squares. For example, if the grid was:

```
ARXX
DCXX
EXXX
XXXX
```

this solution would allow the word, "ARCADE" (0,0) → (0,1) → (1,1) → (0,0) → (1,0) → (2,0). The actual game implemented for the assignment would mark (0,0) as used after the first letter and not allow that grid square's use as the fourth letter in the word as well.

Grading: 5 pts, mostly all or nothing, no explanation necessary. Full credit to any response along the lines of "allows the use of the same tile/grid square more than once". If you feel that an incorrect response deserves partial credit, give what you think it's worth.

6) (10 pts) If we were to remove the line

```
if (!isPrefix(prefix, words)) return;
```

from the solveIt function in the posted solution to program #6 (boggle), then the run time of it would be prohibitive. Clearly explain why this line of code greatly speeds up the run time of the algorithm. Will it speed up the algorithm equally for all input grids? (Assume the input dictionary is the same as the one posted with the solution.) Why or why not?

Solution

There are an exponential number of sequences of strings that can be found in a Boggle grid. Without this line, every possible string that could be formed from the boggle grid would be tried, creating a very large run-time (maybe as large as 3 or 4 raised to the 16^{th}, though the exact time is hard to judge since depending on where you are in the grid, there are more or fewer choices of where to proceed). But, English only has relatively few words compared to the number of possible strings in a boggle grid. By the pigeonhole principle it's relatively easy to show that a vast majority of the strings that you can form in the Boggle grid are not even prefixes to any valid English word. Thus, this single line of code stops any search for strings that are "doomed" to fail because they start with letters that no English word starts with. The number of strings explored can't exceed the number of prefixes in the English language, which a current day computer can easily explore in about a second or so. Finally, the speed up will not be equal for all input grids. Grid with "more bad letters" or fewer short valid prefixes will run faster than grids that have lots of valid prefixes. Here is a grid that will run really fast:

```
XQRT
WZPY
GKKW
QTHV
```

Here is a more fun grid for playing the game and one that would take longer for the posted solution to solve:

```
STRE
HTAE
RAIN
EWHE
```

This is because many prefixes must be explored (due to common letters and the consonant/vowel structure found in this grid) before they get cut off with that line of code.

Grading: The key here is that without that line, all possible strings that can be formed in the grid get "tried" so to speak and the number of these strings is very large. This observation is worth 6 pts. 2 pts are for explaining how the limit on the number of English prefixes makes the run time manageable. 2 pts for explaining why different grids see different speedups.